Science of Computer Programming 97 (2015) 157–162

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Verifying pointer programs using graph grammars ☆

Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, Thomas Noll *

Software Modeling and Verification Group, RWTH Aachen University, Aachen, Germany

HIGHLIGHTS

- We propose to employ graph grammars for modelling dynamic data structures.
- Typical examples include lists, trees and combinations thereof.
- We show how to obtain finite state-space abstractions of heap-manipulating programs.
- This enables the application of standard model checking techniques.
- We report on the automated verification of a stackless tree traversal algorithm.

ARTICLE INFO

Article history:
Received 27 September 2013
Accepted 5 November 2013
Available online 13 November 2013

Keywords:
Hyperedge replacement grammars
Java bytecode
Dynamic data structures
Verification

ABSTRACT

This paper argues that graph grammars naturally model dynamic data structures such as lists, trees and combinations thereof. These grammars can be exploited to obtain finite abstractions of pointer-manipulating programs, thus enabling model checking. Experimental results for verifying Lindstrom's variant of the Deutsch–Schorr–Waite tree traversal algorithm illustrate this.

1. Introduction

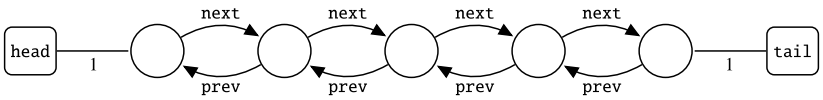
Software heavily relies on pointers. Understanding pointers is thus indispensable so as to understand software. Prominent examples of pointer usage are data structures such as doubly-linked lists, nested lists, trees, and so forth. They occur in device drivers, operating systems, and all kinds of application software. Pointers are also abundantly present in object-oriented software such as Java collections, albeit in the somewhat implicit form of object references. The usage of pointers is error-prone. Some authors claim that the incorrect use of pointers is one of the most common source of bugs in software [1]. Typical problems are dereferencing of null pointers[1] and the creation of memory leaks. Destructive updates through assignment turn out to be a challenge – also referred to as the "complexity of pointer swing" [2]. Automated techniques to find such errors are of great assistance to software engineers. This brief note advocates the usage of graph grammars to facilitate this. We informally describe our approach and present its practical applicability. Important alternative approaches are separation logic [3], shape analysis [4,5], and regular tree automata [6].

2. A graph grammar approach to pointer programs

Heaps as graphs. The starting point of our approach is to consider heaps as graphs. A sample graph for a doubly-linked list with five elements is:

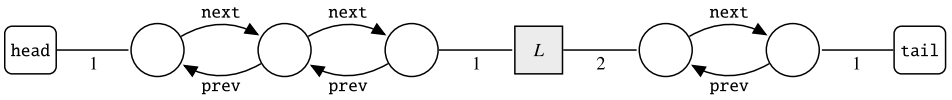
☆ This research has partially been funded by EU FP7 project CARP (Correct and Efficient Accelerator Programming), http://www.carpproject.eu.
* Corresponding author.
[1] In 2009, Hoare considered the introduction of null-references as his "billion dollar mistake", see http://en.wikipedia.org/wiki/Nullable_type.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.012



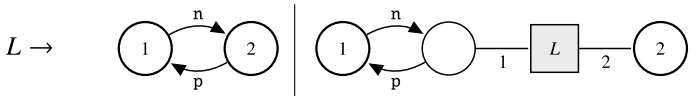
Vertices are represented by circles and stand for heap objects, like cells in a list or nodes of a tree, and edges represent selectors (such as `next` and `prev`) between these objects. Additionally, a mapping of program variables (like `head` and `tail`) to vertices is used to link the program to the heap. Pointer-manipulating operations such as `head := tail.next` or `dispose(head.next)` correspond to graph transformations.

Abstract graphs. Representing a single heap using graphs is rather simple. However, as the number of objects in a heap can be unboundedly large – just consider a loop that creates a new object in each iteration – this would yield graphs with unboundedly many vertices. To obtain finite graph representations we *abstract* subgraphs. Pictorially this looks like

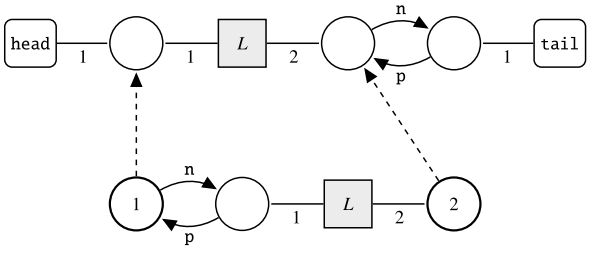


where the edge labelled L represents a doubly-linked list of arbitrary length, i.e., containing zero or more cells. To enable this for general data structures, we slightly generalise our graphs to *hyper*graphs. Rather than having edges connecting two vertices, these graphs feature hyperedges connecting arbitrarily many vertices. The number of vertices connected by a hyperedge is called its *rank* (or its size). In this extended setting, selectors are represented by hyperedges of rank two, which can thus also be interpreted as directed edges, variables by hyperedges of rank one, and abstract subgraphs by hyperedges of arbitrary rank, depending on the data structure that is abstracted from. Hypergraphs are heap configurations that are partially concrete (i.e., precise) and partially abstract (i.e., imprecise). The crucial issue is what to abstract and what not. We return to this issue later on.

Graph grammars. But how do we know which graphs represent the data structures under consideration? That is, how can we obtain all possible heap configurations? This is done by using graph grammars. Graph grammars are similar to string grammars, but manipulate graphs rather than strings. More concretely in the case of hypergraphs, we employ hyperedge replacement grammars [7], shortly referred to as HRGs. Nonterminals such as L in the example above are used to represent abstract fragments of data structures. Terminals represent concrete selectors. Production rules prescribe how nonterminals can be replaced by hypergraphs possibly containing nonterminals. A sample HRG for doubly-linked lists is



where the second production rule recursively adds one list element, whereas the first production rule terminates a derivation. For the sake of brevity, `n` abbreviates `next` and `p` stands for `prev`. The application of a sample rule is illustrated by:



where the second production rule of our grammar is applied to the nonterminal L in the upper graph. To this aim, the "placeholder" nodes labelled by 1 and 2 are mapped to concrete nodes as indicated by the dashed arrows, and L is replaced by the rule's right-hand side. This yields